《计算机通信与网络》

**实验指导手册**

华中科技大学计算机科学与技术学院

二零二四 年 六 月

**目录**

[第1章 实验目标和内容 3](#_Toc178336107)

**[1.1](#_Toc178336108)****[实验目的](#_Toc178336108)** [3](#_Toc178336108)

**[1.2](#_Toc178336109)****[实验环境](#_Toc178336109)** [3](#_Toc178336109)

**[1.3](#_Toc178336110)****[实验要求](#_Toc178336110)** [3](#_Toc178336110)

**[1.4](#_Toc178336111)****[实验内容](#_Toc178336111)** [3](#_Toc178336111)

[第2章 仿真工具介绍 5](#_Toc178336112)

**[2.1](#_Toc178336113)****[OMNeT++介绍](#_Toc178336113)** [5](#_Toc178336113)

**[2.2](#_Toc178336114)****[仿真原理及文件结构](#_Toc178336114)** [5](#_Toc178336114)

[第3章 仿真工具安装及使用 7](#_Toc178336115)

**[3.1](#_Toc178336116)****[OMNeT++安装](#_Toc178336116)** [7](#_Toc178336116)

**[3.2](#_Toc178336117)****[OMNeT++使用](#_Toc178336117)** [7](#_Toc178336117)

**[3.3](#_Toc178336118)****[INET框架](#_Toc178336118)** [19](#_Toc178336118)

[第4章 实验示例代码 28](#_Toc178336119)

**[4.1](#_Toc178336120)****[示例-简单数据收发](#_Toc178336120)** [28](#_Toc178336120)

**[4.2](#_Toc178336121)****[示例-实验一](#_Toc178336121)** [30](#_Toc178336121)

**[4.3](#_Toc178336122)****[示例-实验二](#_Toc178336122)** [34](#_Toc178336122)

**[4.4](#_Toc178336123)****[示例-实验三](#_Toc178336123)** [38](#_Toc178336123)

**[4.5](#_Toc178336124)****[示例-实验四](#_Toc178336124)** [51](#_Toc178336124)

# 实验目标和内容

* 1. **实验目的**

实验一：

* 了解和研究数据链路层协议的工作原理和机制
* 了解和研究动态分配信道
* 学习网络协议原理及应用
* 提高对实际网络问题的分析和解决能力

实验二：

* 了解和研究传输层协议的的工作原理和机制
* 了解和研究因特网上的拥塞控制策略

实验三：

* 了解和研究网络层协议的工作原理及机制
* 掌握路由协议的配置方法
* 掌握路由器及二/三层交换机的配置方法

实验四：

* 学习网络结构的设计
* 学习网络协议原理及应用
* 学习网络的安全管理
* 提高对实际网络问题的分析和解决能力
  1. **实验环境**

OMNet++仿真软件

* 1. **实验要求**
* 利用OMNet++仿真平台完成实验内容。
* 提交实验设计报告纸质档和电子档。
* 基于自己的实验设计报告，通过实验课的上机实验，演示给实验指导教师检查。
  1. **实验内容**

**实验一：**

本实验包括两个级别的内容，具体包括：

* 仿真与测试TCP和UDP协议，比较网络性能，分值为35%。
* 仿真与测试TCP协议中的不同拥塞控制算法（端到端拥塞控制），分值为65%。
* 模拟拥塞环境，根据cwnd等变化过程，理解TCP Tahoe算法、TCP Reno算法、TCP New Reno算法的运行过程，占比为55%。
* 模拟拥塞环境，理解TCP Westwood算法和TCP Vegas算法解决拥塞问题的原理，占比为10%。

**实验二：**

本实验包括三个级别的内容，具体包括：

* + 实现停等式ARQ协议仿真，分值为50%。
    - 给出网络仿真时的动画截图
    - ARQ代码实现并添加注释
    - 截取数据成功传输时和数据传输失败时的信息截图
  + 仿真不同方式的接入控制协议，包括纯ALOHA和时隙ALOHA协议，对比仿真结果并分析，给出相关对比图，分值为30%。
    - 简述纯ALOHA、时隙ALOHA协议工作原理
    - 纯ALOHA与时隙ALOHA仿真场景结果及说明
    - 不同业务强度下纯ALOHA与时隙ALOHA仿真结果比较及分析
  + CSMA协议设计，分值20%。
    - 查找CSMA协议资料，试设计协议，给出协议状态流程图并描述。

**实验三：**

本实验包括三个级别的内容，具体包括：

* 补全OSPF代码，并在路由器配置OSPF协议，使各PC机能互相访问，分值30%
* 补全RIP代码，并在路由器配置RIP协议，使各PC机能互相访问，分值40%
* 补全BGP代码，并在路由器配置BGP协议，使各PC机能互相访问，分值30%

**实验四：**

* 搭建拓扑并测试网络性能，分值为40%。
* 设计一个网络拓扑：该学校有5个学院，1个图书馆，3个学生宿舍，每个学院、图书馆和学生宿舍皆拥有2台主机，学院之间可以相互访问，学生宿舍之间可以相互访问，但学院和学生宿舍之间不能相互访问，学院和学生宿舍皆可访问图书馆。
* 测试网络可用性以及不同网络协议下的网络性能

# 仿真工具介绍

通信系统协议实现很复杂，实现的可靠性和有效性的验证也就复杂。实验方法的局限性

在于成本很高，运用起来不灵活。而仿真方法在很大程度上可以弥补前者的不足，因此得到

广泛应用。仿真工具可以根据需要设计所需的网络模型，用相对较少的时间和费用了解网络

在不同条件下的各种特性，获取网络研究的丰富有效的数据。提供了一个方便、高效的验证

和分析方法，因此网络仿真技术在现代通信网络设计和研究中的作用正变得越来越大。

网络仿真是很好的测试、评估和验证手段之一。当前主流网络仿真软件包括 Opnet，NS2，

OMNet++等。

* 1. **OMNeT++介绍**

OMNeT++是一个免费的、开源的多协议网络仿真软件，在网络仿真领域中占有十分重

要的地位。OMNeT++英文全称是 Objective Modular Network Testbed in C++，是近年来在科学和工业领域里逐渐流行的一种基于组件的模块化的开放的网络仿真平台。OMNeT++作为离散事件仿真器，具备强大完善的图形界面接口。

可以用于无线通信网络和有线通信网络建模、协议仿真建模、排队网络建模、多处理器和分布式硬件系统建模、硬件体系结构验证和评估复杂软件系统多方面的性能等方面。总的来说，它可以用于任何使用离散时间方法的系统仿真和建模，并且其可以方便地映射为依靠交换信息进行的通信的实体。

OMNeT++模型由模块组成，模块之间通过消息传递方式进行通信，OMNeT++的整个模型称为network，模型结构通过NED语言描述，Messages代表网络中的帧或包、排队网络中的工作或者客户，Gates为模块的输入输出接口，Connection（link）用于对物理链路建模，channel具有参数：数据率（data rate）、传播延时（propagation delay）、误码率（bit error rate）、误包率（packet error rate），Module也有参数，在NED文件中指定，或者在配置文件omnetpp.ini中指定。

* 1. **仿真原理及文件结构**

OMNeT++本身并不是所有现实系统的模拟器，但它确实为实现仿真提供了基础底层结

构和工具。这种基础底层结构的基本成分之一是一种用于仿真模型的组件体系结构，模型由可重复使用的元件（即模块）组成。写好的模块可以重复使用，并且能够以各种方式组合。

组成部分：

1. 网络拓扑描述文件：由 NED 语言编写的网络拓扑，便用参数，门，信道链接等来描述模块；

2. 消息定义文件：OMNET++本身提供的消息类型具备一些简单参数，用户还可以根据具体要求通过消息定义文件定义消息成员变量；

3. 简单模块源：简单模块的行为定义文件，包括 C++编写的\*.cc 文件和\*.h；

4. 仿真内核：OMNET 提供的仿真类库代码；

5. 用户接口：该接口用于仿真运行时的测试，演示等工作。

仿真流程：

将系统映射到相互通信的模块体系中，创建模型（模块可以嵌套，多个模块可以组成一

个复合模块）

1. 用 NED(Network Description)语言定义模型的结构,使用 omnet IDE 编辑

2. 利用 omnet 内置内核及类库，采用 c++编译生成模型的活动组件

3. 提供一个拥有配置和参数的 omnetpp.ini 文件给模型（一个配置文件可以用不同的参数来描述若干个仿真过程）

4. 构建仿真程序并运行（代码链接到 OMNet++的仿真内核及其提供的一个用户接口：命令行和交互式接口或图形化接口）

5. 仿真结果将写入输出向量和输出标量文件中，使用 IDE 中提供的分析工具来进行可视化。输出结果是普通的文本，也能用 R，Matlab ，Python或其它工具来进行绘图。

# 仿真工具安装及使用

* 1. **OMNeT++安装**
     1. **下载源文件**

1.访问https://omnetpp.org/download/old，下拉找到OMNeT++ 5.5.1，选择中间的WINDOWS版本，点击下载按钮下载文件“omnetpp-5.5.1-src-windows.zip”。

2. OMNeT++的安装路径需要是英文路径，且路径名不能有空格，创建并进入OMNeT++路径，将第一步下载完成的压缩包解压到该文件夹下，并将解压后的文件夹命名为“omnetpp-5.5.1”。

* + 1. **编译**

1.双击“omnetpp-5.5.1”文件夹下的mingwenv.cmd文件，打开之后会提示“按任意键继续”，点击回车会自动进行文件解压。

2.解压完成后会弹出一个命令行窗口，输入“./configure”回车让其自动执行，配置和检查相关环境，执行结束后将出现“Your PATH contains XXX. Good!”出现。

3.继续输入“make”回车让其自动执行，执行完成会显示“Now you can type “omnetpp” to start the IDE”。

* + 1. **测试安装结果**

1.安装完成后，输入“cd samples/dyna ” 和“./dyna”来测试是否安装成功，如果出现GUI界面说明安装成功。

2.安装完成后，输入”omnetpp”可打开OMNeT++的集成开发环境。

* + 1. **安装问题排查**

1.如遇到Qt5Core.dll找不到的问题，请注意，**运行omnetpp等命令时，均需要在运行mingwenv.cmd后弹出的命令行窗口中输入执行，而非系统的终端窗口**。

2.如遇到make过程失败的问题，请尝试在其他平台下安装。

3.在WSL、Linux、MacOS下安装请查阅参考文档-InstallGuide.pdf。

* 1. **OMNeT++使用**

模型模块中包含算法，仿真对象由C++类描述，OMNeT++模型包含如下几个部分：

（1）NED语言拓扑描述，.ned文件；

（2）消息定义，.msg文件；

（3）简单模块源代码，.h/.cc文件。

仿真系统包含的组件：

（1）仿真内核：管理仿真和仿真类库

（2）用户接口：用于仿真执行，用于调试、显示、批处理仿真执行

* + 1. **NED语言**

NED代表网络描述，是用户描述仿真结构的语言。OMNeT++中，NED可以声明简单模块、连接以及它们组合的复合模块等等。NED语言的特征：继承性、基于组件、接口、层级、包结构、内部类型、元数据注解（属性）。NED语言有树形代表的等价形式，可以序列化为XML。NED语言表示真实的通信网，通信网包含很多节点，每个节点有一个运行的应用程序，应用程序随机发送数据包，节点也是路由，假设应用程序时数据包基础的通信，所以传输层细节可以不用考虑。网络的建模方法为：首先，建立一个network；然后，定义一个网络，引入信道Channel，建立App、Rounting、Queue等模块，再建立节点复合模块，最后所有对象组合在一起。

* 定义一个网络：

network Network

{

submodules:

node1:Node;

node2:Node;

node3:Node;

...

connections:

node1.port++<-->{datarate=100Mbps;} <-->node2.port++;

node2.port++<-->{datarate=100Mbps;} <-->node4.port++;

node4.port++<-->{datarate=100Mbps;} <-->node6.port++;

...

}

Network的类型是network，node1、node2...的类型为Node，Node在下面定义。然后定义节点之间的连接，连接点称为gate，port++的意思是将一个gate加入到port[]中，{datarate=100Mbps;} 代表数据率为100Mbps

* 定义信道

每个连接重复写数据率是很麻烦的，所以，创建一个新的信道类型封装数据率设置

network Network

{

types:

channel c extends ned.DatarateChannel{

datarate=100Mbps;

}

submodules:

node1:Node;

node2:Node;

node3:Node;

...

connections:

node1.port++<-->C<-->node2.port++

...

}

* 简单模块定义

定义流量生成器、路由、队列包等类型

simple App

{

parameters:

int destAddress;

...

@display(“i=block/browser”);

gates:

input in;

output out;

}

simple Routing{...}

simple Queue{...}

模块App、Routing、Queue名以大写开头，以@开头的单词称为“属性”。

定义Node复合模块，将App、Routing和Queue组成Node模块

module Node

{

parameters:

int address;

@display(“i=misc/node\_vs,gold”);

gates:

inout port[];

submodules:

app:App;

routing:Routing;

queue[sizeof(port)]:Queue;

connections:

routing.localOut-->app.in;

routing.localIn<--app.out;

for i=0..sizeof(port)-1{

routing.out[i]-->queue[i].in;

routing.in[i]<--queue[i].out;

queue[i].line<-->port[i];

}

}

* 复合模块的定义

复合模块将其他模块组成一个单元

module Host

{

types:

..

parameters:

..

gates:

...

submodules:

..

connections:

..

}

组装复合模块的例子：

module WirelessHostBase

{

gates:

input radioIn;

submodules:

tcp: TCP;

ip: IP;

wlan: Ieee80211;

connections:

tcp.ipOut --> ip.tcpIn;

tcp.ipIn <-- ip.tcpOut;

ip.nicOut++ --> wlan.ipIn;

ip.nicIn++ <-- wlan.ipOut;

wlan.radioIn <-- radioIn;

}

module WirelessHost extends WirelessHostBase

{

submodules:

webAgent: WebAgent;

connections:

webAgent.tcpOut --> tcp.appIn++;

webAgent.tcpIn <-- tcp.appOut++;

}

module DesktopHost extends WirelessHost

{

gates:

inout ethg;

submodules:

eth: EthernetNic;

connections:

ip.nicOut++ --> eth.ipIn;

ip.nicIn++ <-- eth.ipOut;

eth.phy <--> ethg;

}

* 信道channels

信道封装与连接有关的参数和行为，信道后边有C++类定义，默认的类名是NED中的类型名；如果有@class和@namespace属性，默认的类名可以不是NED中的类型名

系统内置预定义的信道：ned.IdealChannel,ned.DelayChannel,ned.DatarateChannel。如果引入ned包import ned.\*，则不需要写前面的ned.。IdealChannel没有任何参数，DelayChannel有2个参数：delay、disabled，DatarateChannel的参数datarate、ber、per。

* 创建一个新的信道：

channel Ethernet100 extends ned.DatarateChannel

{

datarate=100Mbps;

delay=100us;

ber=1e-10;

}

* 通过子类化来添加参数和属性，也可以修改已有的属性

channel DatarateChannel2 extends ned.DatarateChannel

{

double distance @unit(m);

delay = this.distance / 200000km\*1s;

}

* 添加cost属性

channel Backbone extends ned.DatarateChannel

{

@backbone;

double cost = default(1);

}

* 参数

参数是模块的变量，用于构建拓扑结构，作为C++代码的输入，参数的类型有：doubel，int，bool，string，xml。也可以声明为volatile，单位可以使用@unit属性来指定，参数可以从NED文件或者配置文件omnetpp.ini中读取值。

simple App

{

parameters:

string protocol; // protocol to use: "UDP" / "IP" / "ICMP" / ...

int destAddress; // destination address

volatile double sendInterval @unit(s) = default(exponential(1s));// time between generating packets

volatile int packetLength @unit(byte) = default(100B);// length of one packet

volatile int timeToLive = default(32);// maximum number of network hops to survive

gates:

input in;

output out;

}

上面的模块有5个参数，其中三个有默认值，参数赋值有几个途径：从NED代码赋值、从配置文件omnetpp.ini、从用户交互赋值。NED赋值的几个位置：在继承的子类中、NED类型实例化的子模块和连接定义中、包含子模块或者连接的网络和复合模块中。

* 表达式

NED语言表达式和C语法类似，二进制异或#、逻辑异或##、^是指数、+是串联接，函数：fabs(),toUpper(),uniform(),erlang\_k()，表达式可以使用模块参数、门向量、模块向量、当前向量索引等来引用。

* 在模拟中用于生成随机值的函数：

uniform(a, b)：生成 [a, b] 之间的均匀分布随机数。

normal(mean, stddev)：生成以 mean 为均值、stddev 为标准差的正态分布随机数。

exponential(mean)：生成具有指定均值的指数分布随机数。

truncnormal(mean, stddev)：生成截断的正态分布随机数（用于限制在某个区间内）。

pareto(scale, shape)：生成帕累托分布随机数。

* 单位

同过添加@unit属性可以为参数设置相关的度量单位

simple App

{

parameters:

volatile double sendInterval @unit(s) = default(exponential(350ms));

volatile int packetLength @unit(byte) = default(4KiB);

...

}

* XML参数

模块需要复杂数据输入

OMNeT++包含XML内置支持。读取DTD验证文件，缓存文件，使用XPath子集选择文档部分，提供DOM形式对象树的内容。

使用类型xml，通过xmldoc()函数将xml类型参数指向一个xml文件

simple TrafGen {

parameters:

xml profile;

gates:

output out;

}

module Node {

submodules:

trafGen1 : TrafGen {

profile = xmldoc("data.xml");

}

...

}

* 门

门是模块的连接点，有三种类型：input、output、inout，一般NED要求所有门都有连接，为了放松这个条件，可以使用@loose来注解

simple GridNode {

gates:

inout neighbour[4] @loose;

}

如果输入门收到的消息是sendDirect()函数发出的，使用注解@directIn

simple WirelessNode {

gates:

input radioIn @directIn;

}

* 元数据注解（属性）

NED属性是元数据注解，能够添加到模块、参数、门、连接、NED文件、包和NED任何元素上。

提过的属性：@display，@class，@namespace，@unit，@prompt，@loose，@directIn

使用属性，可以让NED元素挂接额外的信息，一些属性由NED解释，一些由仿真内核解释；其他属性用于仿真模型以内。属性作用于类型。

@namespace(foo); // file property

module Example

{

parameters:

@node; // module property

@display("i=device/pc"); // module property

int a @unit(s) = default(1); // parameter property

gates:

output out @loose @labels(pk); // gate properties

submodules:

src: Source {

parameters:

@display("p=150,100"); // submodule property

count @prompt("Enter count:"); // adding a property to a parameter

gates:

out[] @loose; // adding a property to a gate

}

...

connections:

src.out++ --> { @display("ls=green,2"); } --> sink1.in; // connection prop.

src.out++ --> Channel { @display("ls=green,2"); } --> sink2.in;

}

* + 1. **.cc文件**
* 定义简单模块类型

简单模块类型就是从cSimpleModule子类化得到的C++类，通过重定义一个或者多个虚函数来定义行为。**该类必须通过Define\_Module()宏在OMNeT++中注册， Define\_Module()宏放在.cc或者.cpp文件中**。

// file: HelloModule.cc

#include <omnetpp.h>

class HelloModule : public cSimpleModule{

protected:

virtual void initialize();

virtual void handleMessage(cMessage\*msg);

};

Define\_Module(HelloModule); // register module class with OMNeT++

void HelloModule::initialize(){ev << "Hello World!\n";}

void HelloModule::handleMessage(cMessage\*msg){

delete msg; // just discard everything we receive

}

为了能在NED文件中使用上述建立的模块类型，需要一个相关的NED声明：

simple HelloModule{

gates:

input in;

}

构造函数：简单模块是仿真内核直接实例化，而不是用户进行的。这就意味着不能随意写构造函数，签名必须由仿真内核指定。幸运地是，这条协议很简单：构造函数必须是public，且没有参数

publc:

HelloModule();

cSimpleModule本身有2个构造函数：cSimpleModule(); cSimpleModule(size\_t stacksize)用于接收协同程序的堆栈尺寸。前者用于handleMessage，后者用于activity()

HelloModule::HelloModule() {...}

HelloModule::HelloModule() : cSimpleModule() {...}//使用handleMessage

HelloModule::HelloModule() : cSimpleModule(16384) {...}//选择activity

构造函数、析构函数、initialize、finish四个函数有下面使用惯例：

构造函数：将模块的成员变量指针设为NULL；其他初始化工作放到initialize中

initialize：进行初始化任务。读模块参数、初始化类变量、使用new分配动态数据结构、分配和初始化自定义的消息、计时器。

finish：记录统计。不要delete任何对象，不要取消计时器

析构函数：delete所有使用new创建的对象，使用cancelAndDelete删除自消息（计时器）

调用顺序：

最初的事件（starter messages）放入FES → 所有模块的initialize函数 → 进入事件循环 → 第一个事件被处理 →... → 结束事件循环 → 所有模块的finish函数

复合模块的initialize → 子模块initialize

如果仿真器因为运行时错误而终止，finish函数不会被调用

finish调用方式和initialize相反。子模块finish → 复合模块finish

* cSimpleModule添加功能——handleMessage()

handleMessage函数被每个事件调用

想法就是，每个事件发生（消息到达）的时候调用一个用户定义的函数。handleMessage是cSimpleModule的虚成员函数，函数体默认是空的，在子类中用户需要重新定义，并添加消息处理的代码。模块每个消息到达的时候，handleMessage函数会被调用。仿真器中的事件循环同时处理activity和handMessage函数，对应于如下的伪代码：

带有handleMessage的模块不是自动开始的：

仿真内核只为带有activity的模块创建开始消息starter message。这就意味着，必须从initialize函数调度自消息self-messages，才可以让handMessage模块自己开始工作，不是从其他模块收到第一个消息。

* handleMessage编程

为了在简单模块中使用handleMessage机制，应该为模块指定堆栈大小为0。

在handleMessage中能使用的函数有send、scheduleAt、cancelEvent

send()这一类的函数：发送消息给其他模块

scheduleAt():调度一个事件（模块“发送消息给自己”）

cancelEvent():删除由schedleAt函数调度的事件

不能在handleMessage中使用receive和wait函数，因为它们天然是协同程序基础的，用于activity想保存的每个数据信息必须在模块类中添加为数据成员，不能够作为handleMessage的本地变量存储。同时，也不能存放在静态变量中，因为静态变量在类的所有实例中共享。

加入模块类中的数据成员变量包括如下的情形：状态、属于模块状态的其他变量（重试次数、包队列等）、计算/获取的一次值保存（模块参数值、门索引、路由信息等）、统计集合的变量/对象。这些变量都可以在initialize函数中初始化。

initialize的另一个任务是初始化首次触发handleMessage的最初时间，首次调用之后，handleMessage必须注意为自己调度事件以保持“链”不会断裂。

* activity()处理过程描述

使用activity，可以想编写操作系统进程或线程一样对简单模块进行编程，可以等待输入消息，可以停止执行一段时间，当activity退出时，模块终止。

activity包含一个无限循环，函数体中可以使用的重要函数如下：

receive()-收到消息（事件）

wait()-停止执行一段时间

send()-发送消息给其他模块

scheduleAt()-调度一个事件

cancelEvent()-删除使用scheduleAt调度的事件

end()-结束这个模块的执行

使用activity的模块需要starter message来“启动”。在仿真开始之前，这些开始消息自动插入FES中，甚至比initialize函数调用还早。

使用activity的时候，程序员需要指定堆栈大小，16k或者32k通常是一个很好的选择。

因为activity中的局部变量在整个事件过程中都保留着，所有刻意将任何信息存放在activity函数中。然而，如果需要在仿真结束的时候写入统计量，就需要finish函数。这样，典型的设置如下伪代码所示：

class MySimpleModule...

{

...

variables for statistics collection

activity();

finish();

};

MySimpleModule::activity()

{

declare local vars and initialize them

initialize statistics collection variables

while(true)

{...}

}

MySimpleModule::finish()

{

record statistics into file

}

* 访问门和连接

门对象：cGate对象代表模块门，使用cModule的gate方法，利用名字查询门cGate \*outGate=gate(“out”);

对于门向量，访问向量中单独的门时，需要在调用gate函数时加一个索引参数，索引位于0~size-1之间

另外，可以通过门id来访问门

添加和删除门addGate(name,type,isvector=false)、deleteGate(name)

简单模块门只有一个连接，复合模块门需要模块的输入和输出门都使用。一系列的连接称为连接路径或者路径。一条路径一般开始于一个简单模块的输出门，终止于输入门。每个cGate对象包含前一个门和后一个门的指针，所以路径可以看作一个双链表。

同一个连接关联的信道对象可以通过存储在连接源门的指针进行访问。指针通过getChannel方法返回cChannel \*channel=gate->getChannel();

如果有信道指针，就可以使用getSourceGate()方法得到源门cGate \*gate=channel->getSourceGate();

cChannel只是信道的抽象基类，为了访问信道的细节，必须将指针转换为实际的信道类型。

* 发送消息

在一个抽象层次上，一个OMNeT++仿真模型是通过消息传递方式通信的简单模块的集合。简单模块的关键在于消息的创建、发送、接收、存储、修改、调度和销毁，剩下的任务就是收集统计的数据。

OMNeT++中消息是cMessage类或者其子类的一个实例。网络包用cPackage类代表，该类也是cMessage的一个子类。消息对象使用C++的new操作符创建，使用delete操作符销毁。

一旦创建，消息对象可以通过输出门发送出去：

send(cMessage \*msg, const char \*gateName, int index=0);

send(cMessage \*msg, int gateId);

send(cMessage \*msg, cGate \*gate);

当实现广播或者重传时，可能会用同样的消息进行send操作，这样做是不对的，同一个消息对象不能发送多次。正确的做法是，复制消息对象。

为什么呢？一个消息就像现实世界的真实对象，不能同时在2个地方。一旦发送出去，消息对象就不再属于模块：它会被仿真内核接管，最后传递给目标模块。发送模块甚至不能再引用它的指针。

一旦消息到达目标模块，该模块就有了完全的权限——可以发送、销毁、或者存放以备后用。这也同样适用于调度的消息（属于内核的直到传递回模块）

为了加强这些规则，所有的消息发送函数会检查要发送的消息的实际拥有人。如果消息在另一个模块里，就会得到一个运行时错误：not owner of message

广播消息的实现过程：发送同一个消息的多个拷贝，广播完毕后删除该消息。

for (int i=0; i<n; i++){

cMessage \*copy = msg->dup();

send(copy, "out", i);

}

delete msg;

或者int outGateBaseId = gateBaseId("out");

for (int i=0; i<n; i++)send(i==n-1 ? msg : msg->dup(), outGateBaseId+i);

重传时，只需要保留同一个消息对象的指针，一次一次发送就行了。

// (re)transmit packet:

cMessage \*copy = packet->dup();

send(copy, "out");

不需要重传时就delete packet;

有时模块有必要将消息保留一段时间，然后发送，一般使用sendDelayed函数实现

sendDelayed(cMessage \*msg, double delay, const char \*gateName, int index);

sendDelayed(cMessage \*msg, double delay, int gateId);

sendDelayed(cMessage \*msg, double delay, cGate \*gate);

有时，直接发送消息到另一个模块的输入门是很方便的，为达到这个目的是用sendDirect函数

sendDirect(cMessage \*msg, cModule \*mod, int gateId)

sendDirect(cMessage \*msg, cModule \*mod, const char \*gateName, int index=-1)

sendDirect(cMessage \*msg, cGate \*gate)

直接传送的例子如下：

cModule \*targetModule = getParentModule()->getSubmodule("node2");

sendDirect(new cMessage("msg"), targetModule, "in");

* 接收消息

当消息传送到模块，包的isReceptionStart方法可以被调用来决定其是相应的接收考试还是结束。下面的代码打印收到包的开始和结束时间。

stimtime\_t startTime,endTime;

if(pkg-isReceptionStart()){

//gate was reprogrammed with setDeliverOnReceptionStart(true)

startTime = pkg->getArrivalTime();

endTime= stratrtTime + pkg->getDuration();

} else {

//default case

endTime=pkg->getArrivalTime();

startTime=endTime-pkg->getDuration();

}

ev << “interval: “ << startTime << “..” << endTime << “\n”;

* + 1. **.ini文件**

仿真器的配置和输入数据常常位于omnetpp.ini的配置文件中，文件是面向行的，包括节头线、键值对、声明行，[General]是最顶头的节，ini可指定模块的参数，该文件的书写方式示例如下：

[General]

network = Ipv4hook

\*\*.host\*.numApps = 1

\*\*.host1.app[0].typename = "UdpBasicApp"

\*\*.host1.app[0].destAddresses = "host2"

\*\*.host1.app[0].destPort = 1000

\*\*.host1.app[0].messageLength = 2000B

\*\*.host1.app[0].sendInterval = 1s

\*\*.host1.app[0].startTime = 1s

\*\*.host1.app[0].stopTime = 2.5s

\*\*.host2.app[0].typename = "UdpEchoApp"

\*\*.host2.app[0].localPort = 1000

* 1. **INET框架**

INET框架是OMNeT++仿真环境的开源模型库。它为研究人员和使用通信网络的学生提供协议、代理和其他模型。INET在设计和验证新协议、探索新的或特殊的场景时非常有用。

INET支持通用的通信网络，包括有线、无线、移动、自组织和传感器网络。它包含互联网堆栈模型（TCP、UDP、IPv4、IPv6、OSPF、BGP等）、链路层协议（以太网、PPP、IEEE 802.11、各种传感器MAC协议等）、无线物理层支持、MANET路由协议、DiffServ、带有LDP和RSVP-TE信令的MPLS、若干应用模型以及许多其他协议和组件。它还为节点移动性、高级可视化、网络仿真等提供支持。

其他一些仿真框架以INET为基础，并将其扩展到特定方向，如车辆网络、覆盖/对等网络或LTE。

INET是通过消息传递进行通信的模块的概念构建的。代理和网络协议由组件表示，组件可以自由组合以形成主机、路由器、交换机和其他网络设备。新组件可由用户编程，现有组件已编写，便于理解和修改。

INET得益于OMNeT++提供的基础设施。除了利用OMNeT++仿真内核和库提供的服务（组件模型、参数化、结果记录等）之外，这还意味着可以从OMNeT++ Simulation IDE或命令行开发、组装、参数化和运行模型，并评估其结果。

INET框架由OMNeT++团队为社区维护，利用社区成员提供的补丁和新模型。

* + 1. **安装**

1.OMNeT++IDE下载并安装（推荐）。在第一次启动IDE时出现的对话框中接受安装INET的提议即可安装INET；如果你未选择安装，可单击**菜单栏-帮助-安装仿真模型（Install Simulation Models）**选择INET Framework 4安装。

2.从INET框架网站：<http://inet.omnetpp.org>。IDE始终安装与OMNeT++版本兼容的最新稳定版本。如果需要其他版本，可以从网站下载，网站中还提供了安装说明。

3.GitHub。<http://github.com/inet-framework/>。

* + 1. **框架结构**

INET框架建立在OMNeT++之上，并使用相同的概念：通过消息传递进行通信的模块。主机、路由器、交换机和其他网络设备由OMNeT++复合模块表示。这些复合模块由表示协议、应用程序和其他功能单元的简单模块组装而成。网络也是一个包含主机、路由器和其他模块的OMNeT++复合模块。

INET模块的在线参考文档：<https://doc.omnetpp.org/inet/api-current/neddoc/index.html>

模块大致遵循OSI层的目录结构：

* src/inet/applications/ – 流量生成器和应用模型
* src/inet/transportlayer/ – 传输层协议
* src/inet/networklayer/ – 网络层协议和附件
* src/inet/linklayer/ – 链路层协议和附件
* src/inet/physicallayer/ – 物理层模型
* src/inet/routing/ – 路由协议(internet and ad hoc)
* src/inet/mobility/ – mobility models
* src/inet/power/ – energy consumption modeling
* src/inet/environment/ – model of the physical environment
* src/inet/node/ – 预装配的网络节点模型
* src/inet/visualizer/ – visualization components (2D and 3D)
* src/inet/common/ – miscellaneous utility components

OMNeT++NED语言使用分层包名称。包对应于下的目录，例如，目录src/src/inet/transportlayer/tcpinet.transportlayer.tcp对应于NED包。

对于模块化，INET框架定义了大约80个项目特性（可以作为一个单元禁用的部分代码库）。安装后，并非所有项目功能都在默认设置中启用。可以在IDE的Project ‣ Project Features…对话框中查看可用项目功能列表。如果想了解有关项目功能的更多信息，请参阅《OMNeT++用户指南》。

* + 1. **TCP协议**

TCP协议是Internet上使用最广泛的协议。它提供了从一台计算机上的一个应用程序到另一台计算机中的另一应用程序的可靠、有序的字节流传递。RFC793中描述了基准TCP协议，但其他数十个RFC包含对TCP的修改和扩展。因此，TCP是一个复杂的协议，有时很难看出不同的需求是如何相互作用的。

INET包含TCP协议的三种实现：

* Tcp是主要使用方式，旨在提高可读性、可扩展性和实验性。
* TcpLwip是lwIP（轻量级IP）库的包装器，lwIP是为嵌入式系统设计的广泛使用的开源TCP/IP堆栈。
* TcpNsc包装了网络模拟摇篮（NSC），这是一个允许在网络模拟器中使用真实世界TCP/IP网络堆栈的库。

所有三种模块类型都实现ITcp接口，并通过同一接口与其他层通信，因此它们可以在同一网络中互换和混合。

* **Tcp**

Tcp简单模块是INET框架中Tcp协议的主要实现。Tcp实现一下功能：

* TCP状态机
* 根据系统时钟选择初始序列号。
* 基于窗口的流量控制
* 窗口比例选项
* 持续计时器
* 保持活动计时器
* 传输策略
* 用于重传超时（RTO）计算的RTT测量
* 延迟ACK算法
* Nagle算法
* 糊涂窗口避免
* 时间戳选项
* 拥塞控制算法：Tahoe, Reno, New Reno, Westwood, Vegas, etc.
* 缓慢启动和避免拥堵
* 快速重新传输和快速恢复
* 使用有限传输恢复损失
* 选择性确认（SACK）
* 基于SACK的损失恢复

拥塞控制算法可以用该参数来选择。例如，以下ini文件片段选择TCP Vegas：tcpAlgorithmClass

\*\*.tcp.tcpAlgorithmClass = "TcpVegas"

通过在C++中实现和编写新的拥塞控制方案可以扩展Tcp。

* **TcpLwip**

lwIP是最初由瑞典计算机科学研究所的Adam Dunkels编写的TCP/IP协议套件的轻量级实现。当前开发主页为http://savannah.nongnu.org/projects/lwip/.

该实现以嵌入式设备为目标：它的资源使用量非常有限（可以“使用数十千字节的RAM和大约40千字节的ROM”），并且不需要底层操作系统。

TcpLwip简单模块基于lwIP源的1.3.2版本。

* 延迟ACK
* Nagle算法
* 往返时间估计
* 自适应重传超时
* SWS避免
* 慢启动阈值
* 快重传
* 快速恢复
* persist timer
* keep-alive timer
* **TcpNsc**

Network Simulation Cradle（Network Simulation Cradle，NSC）是一种允许在模拟网络中使用真实TCP/IP网络堆栈的工具。NSC项目由Sam Jansen创建，可在<http://research.wand.net.nz/software/nsc.php>中查看具体内容。NSC目前包含Linux、FreeBSD、OpenBSD和lwIP网络堆栈，但在64位系统上只能构建Linux实现。

要使用TcpNsc模块，应该下载软件包并按照文件中的说明构建。

nsc-0.5.2.tar.bz2<inet\_root>/3rdparty/README

* + 1. **UDP协议**

UDP协议是一种非常简单的数据报传输协议，使网络层的服务对应用程序可用。它只执行端口的分组复用和解复用以及一些基本的错误检测。

Udp简单模块实现UDP协议。有一个模块接口（IUdp）定义了Udp组件的门。在StandardHost节点中，UDP组件可以是实现该接口的任何模块。

**发送UDP数据报：**

如果应用程序想要发送数据报，它可以选择连接到目的地。它发送一条带有UDP\_C\_CONNECT类型和控制信息的消息，其中包含连接的远程地址和端口。UDP协议实际上是无连接的，因此它不会因为连接调用而发送任何数据包。当UDP模块接收到连接请求时，它只需记住目标地址和端口，并将其用作以后发送的默认目标。应用程序可以向同一个套接字发送多个连接命令。UdpConnectCommand

对于发送UDP数据包，应用程序应该在数据包上附加一个控制信息，并将其发送给UDP。控制信息可能包含目标地址和端口。如果控制信息中未指定目标地址或端口，则数据包将发送到连接的目标。UDPSendCommand

Udp模块将应用程序的数据包封装到UdpHeader中，创建适当的IP控制信息，并根据目的地地址通过ipOut或ipv6Out发送。

**接收UDP数据报：**

在接收UDP数据报之前，应用程序应首先“绑定”到给定的UDP端口。这可以通过发送带有控制信息的消息类型UDP\_C\_bind来完成。控制信息包含套接字标识符以及应用程序想要接收UDP数据包的本地地址和端口。地址和端口都是可选的。如果未指定地址，则将具有任何目的地址的UDP数据包传递给应用程序。如果端口为-1，则UDP模块会自动选择一个未使用的端口。localAddress/localPort组合必须是唯一的。

当数据包从网络到达时，首先检查其错误位。UDP组件会丢弃错误的消息。否则，将查找绑定到目标端口的应用程序，并将解封装的数据包传递给该应用程序。如果没有应用程序绑定到目的端口，则会向数据包的源发送ICMP错误。如果套接字已连接，那么只有从连接的远程地址和端口接收的数据包才会被传递到应用程序。

解封装数据包的控制信息包含有关源和目标地址/端口、TTL以及接收数据包的接口卡的标识符的信息。

* + 1. **OSPF协议**

OSPF（Open Shortest Path First，开放式最短路径优先）是一种用于IP网络的路由协议。它使用链路状态路由（LSR）算法，属于内部网关协议（IGP）组，在单个自治系统（AS）内运行。

OspfRouter是一个启用了OSPF协议的路由器。

Ospfv2模块实现OSPF协议版本2。可以使用ospfConfig参数指定的XML文件来配置区域和路由器。网络接口的各种参数也可以在XML文件中指定，或者作为Ospfv2模块的参数。

\*\*.ospf.ospfConfig = xmldoc("ASConfig.xml")

\*\*.ospf.helloInterval = 12s

\*\*.ospf.retransmissionInterval = 6s

<OSPFASConfig>根元素可能包含<Area>和<Router>元素，这些元素具有指定网络接口参数的各种属性。最重要的是，<Area>包含<AddressRange>元素，这些元素列举了应该由协议播发的网络地址。此外，<Router>元素可能包含用于配置各种点对点或广播接口的数据。

<?xml version="1.0"?>

**<OSPFASConfig** xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="OSPF.xsd"**>**

*<!-- Areas -->*

**<Area** id="0.0.0.0"**>**

**<AddressRange** address="H1" mask="H1" status="Advertise" **/>**

**<AddressRange** address="H2" mask="H2" status="Advertise" **/>**

**<AddressRange** address="R1>R2" mask="R1>R2" status="Advertise" **/>**

**<AddressRange** address="R2>R1" mask="R2>R1" status="Advertise" **/>**

**</Area>**

*<!-- Routers -->*

**<Router** name="R1" RFC1583Compatible="true"**>**

**<BroadcastInterface** ifName="eth0" areaID="0.0.0.0" interfaceOutputCost="1" routerPriority="1" **/>**

**<PointToPointInterface** ifName="eth1" areaID="0.0.0.0" interfaceOutputCost="2" **/>**

**</Router>**

**<Router** name="R2" RFC1583Compatible="true"**>**

**<PointToPointInterface** ifName="eth0" areaID="0.0.0.0" interfaceOutputCost="2" **/>**

**<BroadcastInterface** ifName="eth1" areaID="0.0.0.0" interfaceOutputCost="1" routerPriority="2" **/>**

**</Router>**

**</OSPFASConfig>**

* + 1. **RIP协议**

RIP（路由信息协议）是一种使用跳数作为路由度量的距离矢量路由协议。RIP通过限制从源到目的地的路径中允许的跳数来防止路由循环。RIP使用split horizon with poison reverse技术来解决“计数到无穷大”的问题。

Rip模块实现如RFC 2453（RIPv2）和RFC 2080（RIPng）中所规定的距离矢量路由。可以通过将模式参数设置为“RIPv2”或“RIPng”来选择行为。可以使用ripConfig 参数在XML中指定协议配置，例如链路度量和每个接口的操作模式（例如是否在接口上启用RIP，以及是否使用拆分范围）。

以下示例将路由器模块配置为使用RIPv2：

\*\*.hasRip = true

\*\*.mode = "RIPv2"

\*\*.ripConfig = xmldoc("RIPConfig.xml")

配置文件指定每个接口的参数。每个＜interface＞元素配置一个或多个接口；主机、名称、朝向和属性中选择配置的接口（与Ipv4NetworkConfigurator类似）。

* 其他属性：

metric：指定给链接的度量，默认值为1。该值被添加到在该接口上接收的学习路线的度量中。它必须是[1，15]区间内的整数。

mode：界面的模式。

* model属性可以是以下属性之一：

NoRIP：此接口上不发送或接收任何RIP消息。

NoSplitHorizon：无水平分割过滤；将所有路由发送给邻居。

SplitHorizon：不发送下一跳是邻居的路由。

SplitHorizonPisenedReverse（默认）：如果下一跳是邻居，则将路由的度量设置为无穷大。

以下示例将路由器R1和RB之间的链路度量设置为2，而所有其他链路将具有1的度量。

**<RIPConfig>**

**<interface** among="R1 RB" metric="2"**/>**

**<interface** among="R? R?" metric="1"**/>**

**</RIPConfig>**

* + 1. **BGP协议**

边界网关协议（BGP）是一种标准化的外部网关协议，旨在在互联网上的自治系统（AS）之间交换路由和可达性信息。

BgpRouter是一个启用了BGP协议的路由器。

Bgp模块实现Bgp版本4。该模型实现了RFC 4271，但有一些限制。可以在bgpConfig参数中指定的XML文件中配置自治系统和规则。

\*\*.bgpConfig = xmldoc("BGPConfig.xml")

配置文件可能包含顶层的<TimerParams>、<AS>和Session元素。

* ＜TimerParams＞：允许为路由器指定各种定时参数。
* <AS>：定义要应用的自治系统、路由器和规则。
* ＜Session＞：指定边缘路由器之间的打开会话。它必须正好包含两个<Router exterAddr="x.x.x.x"/>元素。

**<BGPConfig** xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="BGP.xsd"**>**

**<TimerParams>**

**<connectRetryTime>** 120 **</connectRetryTime>**

**<holdTime>** 180 **</holdTime>**

**<keepAliveTime>** 60 **</keepAliveTime>**

**<startDelay>** 15 **</startDelay>**

**</TimerParams>**

**<AS** id="60111"**>**

**<Router** interAddr="172.1.10.255"**/>** *<!--Router A1-->*

**<Router** interAddr="172.1.20.255"**/>** *<!--Router A2-->*

**</AS>**

**<AS** id="60222"**>**

**<Router** interAddr="172.10.4.255"**/>** *<!--Router B-->*

**</AS>**

**<AS** id="60333"**>**

**<Router** interAddr="172.13.1.255"**/>** *<!--Router C1-->*

**<Router** interAddr="172.13.2.255"**/>** *<!--Router C2-->*

**<Router** interAddr="172.13.3.255"**/>** *<!--Router C3-->*

**<Router** interAddr="172.13.4.255"**/>** *<!--Router C4-->*

**<DenyRouteOUT** Address="172.10.8.0" Netmask="255.255.255.0"**/>**

**<DenyASOUT>** 60111 **</DenyASOUT>**

**</AS>**

**<Session** id="1"**>**

**<Router** exterAddr="10.10.10.1" **>** **</Router>** *<!--Router A1-->*

**<Router** exterAddr="10.10.10.2" **>** **</Router>** *<!--Router C1-->*

**</Session>**

**<Session** id="2"**>**

**<Router** exterAddr="10.10.20.1" **>** **</Router>** *<!--Router A2-->*

**<Router** exterAddr="10.10.20.2" **>** **</Router>** *<!--Router B-->*

**</Session>**

**<Session** id="3"**>**

**<Router** exterAddr="10.10.30.1" **>** **</Router>** *<!--Router B-->*

**<Router** exterAddr="10.10.30.2" **>** **</Router>** *<!--Router C2-->*

**</Session>**

**</BGPConfig>**

在<AS>元素内部，可以单独指定各种规则：

* DenyRoute：拒绝IN和OUT流量中的路由（必须指定Address和Netmask属性。）
* DenyRouteIN:拒绝IN流量中的路由（必须指定Address和Netmask属性。）
* DenyRouteOUT:拒绝OUT流量中的路由（必须指定Address和Netmask属性。）
* DenyAS：拒绝AS在IN和OUT流量中学习的路由（必须将AS id指定为元素的主体。）
* DenyASIN：拒绝AS在IN流量中学习的路由（必须将AS id指定为元素的主体。）
* DenyASOUT：拒绝AS在OUT流量中学习的路由（必须将AS id指定为元素的主体。）

# 实验示例代码

* 1. **示例-简单数据收发**
     1. **新建OMNeT++项目**

1.运行mingwenv.cmd，在打开的命令行窗口中输入omnetpp命令打开IDE

2.打开simulation视图：Windows-->perspective-->open perspective-->simulation

3.新建项目

单击菜单栏-文件（File）-新建（new）-创建OMNeT++项目（OMNeT++ Project）。

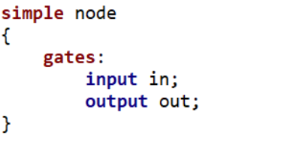
输入项目名称后，点击Finish创建完成

* + 1. **新建拓扑**

1. 新建简单子模块node工程

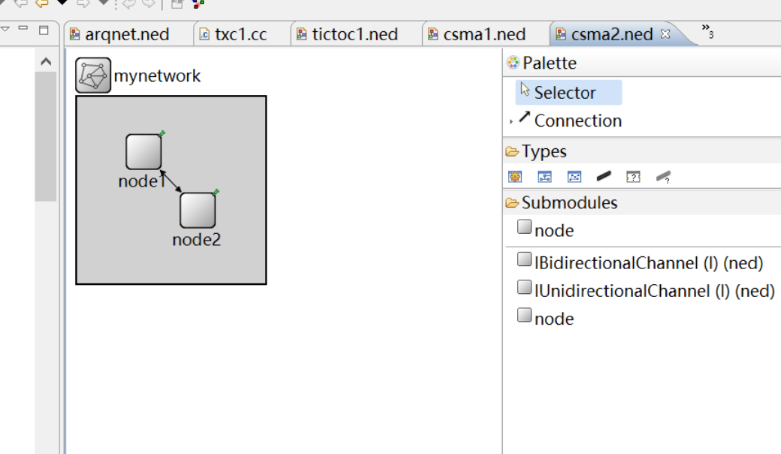
右键项目-->新建（new） -->Network Description File（NED），新建一个 NED 文件，命名为 xxx.ned，选择Empty NED File。

建立新的子模块可以通过图形界面（在项目列表中双击xxx.ned-->点击窗口下方的Design视图）的方式直接拉取，也可以通过代码（在项目列表中双击xxx.ned-->点击窗口下方的Source视图）的方式。如果无Design-Source按钮，请右键NED文件使用NED Editor打开而非Text Editor。打开后代码如下：

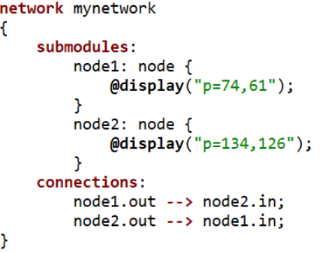


2.新建网络拓扑

新建一个 NED 文件，命名为 xxx.ned，选择NED file with one item，选择Inital Content为Network。在Design视图下的Submodules子视图中，选择刚才创建的简单模块，在拓扑中添加两个节点，并建立他们之间的双向连接。在网络中添加 2 个实体子模块（必须先添加网络的灰框，再添加单个节点）。如图所示：



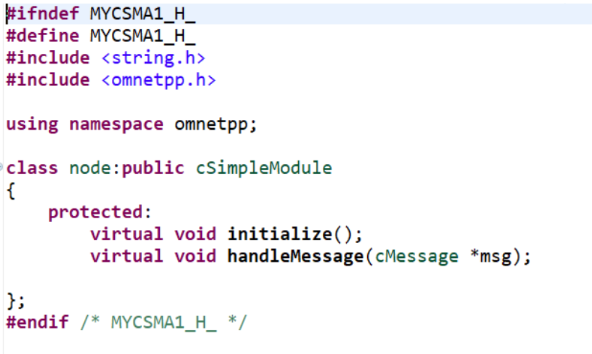
代码如下：



* + 1. **收发包模块**

在项目文件夹上右键，新建源文件 New->SourceFile，创建.h 文件和.cpp 文件让网络工作起来。

其中，.h文件如下：



其中，.cpp文件如下：



* + 1. **创建.ini文件并运行仿真**

1. 右键项目，单击新建（new）-选择initialization file，拓扑与之前建立的拓扑名一致（如示例中的mynetwork），得到omnetpp.ini文件。

2. 添加仿真时间，注意填写单位s，此处仿真时间设置10s

[General]

network = mynetwork

sim-time-limit = 10s

simtime-resolution = ps

3.右键项目，选择build project，完成编译。

4.右键单击ini文件或菜单栏/工具栏中的运行按钮Run as OMNet++ Simulation，等待加载完成后会出现仿真界面，所呈现的网络拓扑与设置一致。

5.单击仿真窗口工具栏中的单步执行、运行、快速运行、停止、重置等按钮查看仿真结果，建议掌握相关快捷键。在控制台输出窗口的按钮列表中，可以切换流量列表/日志样式，并提供筛选、查找等功能。在Preferences中可以调整日志级别等配置。

* 1. **示例-实验一**
     1. **TCP/UDP模块**

在新创建的项目使用INET内置的模块前，需要右键项目-属性（Properties）-项目引用（Project References）-勾选inet，然后在ned文件中导入相应的类，类的路径查看inet/src或在线参考文档。

1.添加拓扑，代码如下：

**import** inet.common.misc.ThruputMeteringChannel;

**import** inet.networklayer.configurator.ipv4.Ipv4NetworkConfigurator;

**import** inet.node.inet.Router;

**import** inet.node.inet.StandardHost;

**network** TCPUDP

{

**parameters**:

**int** n; *// n for Tcp client number.*

**types**:

**channel** ethernetline **extends** ThruputMeteringChannel

{

delay=1us;

datarate=10kbps;

thruputDisplayFormat = "p-b-u";

}

**submodules**:

configurator: Ipv4NetworkConfigurator {

**parameters**:

**@display**("p=100,100;is=s");

}

router1: Router {

**@display**("p=400,250");

}

router2: Router {

**@display**("p=550,250");

}

router3: Router {

**@display**("p=700,250");

}

client[n]: StandardHost {

**parameters**:

**@display**("i=device/laptop;p=250,100,col,100");

}

clientUdp: StandardHost {

**parameters**:

**@display**("i=device/laptop;p=240,600,col,100");

}

server: StandardHost {

**parameters**:

**@display**("i=device/server;p=850,150");

}

serverUdp: StandardHost {

**parameters**:

**@display**("i=device/server;p=850,250");

}

**connections**:

**for** i=0..n-1 {

client[i].pppg++ **<-->** ethernetline **<-->** router1.pppg++;

}

clientUdp.pppg++ **<-->** ethernetline **<-->** router1.pppg++;

router1.pppg++ **<-->** ethernetline **<-->** router2.pppg++;

router2.pppg++ **<-->** ethernetline **<-->** router3.pppg++;

router3.pppg++ **<-->** ethernetline **<-->** server.pppg++;

router3.pppg++ **<-->** ethernetline **<-->** serverUdp.pppg++;

}

2.添加ini文件，相关配置如下：

[General]

network = TCPUDP

*# udp client*

\*\*.clientUdp.numApps = 1 *# the number of applications*

\*\*.clientUdp.app[\*].typename = "UdpBasicApp" *# inet.applications.udpapp.UdpBasicApp*

\*\*.clientUdp.app[\*].destAddresses = "serverUdp"

\*\*.clientUdp.app[\*].destPort = 1000

\*\*.clientUdp.app[\*].messageLength = 100B

\*\*.clientUdp.app[\*].startTime = 1s

\*\*.clientUdp.app[\*].sendInterval = 1s

*# udp server*

\*\*.serverUdp.numApps = 1

\*\*.serverUdp.app[\*].typename = "UdpEchoApp" *# inet.applications.udpapp.UdpEchoApp*

\*\*.serverUdp.app[\*].localPort = 1000

*# tcp client*

\*.n = 5

\*\*.client[\*].numApps = 1

\*\*.client[\*].tcp.typename = "Tcp"

\*\*.client[\*].app[0].typename = "TcpBasicClientApp" *# inet.applications.tcpapp.TcpBasicClientApp*

\*\*.client[\*].app[0].localAddress = ""

\*\*.client[\*].app[0].localPort = -1

\*\*.client[\*].app[0].connectAddress = "server"

\*\*.client[\*].app[0].connectPort = 80

\*\*.client[\*].app[0].startTime = 1s *# time first session begins*

\*\*.client[\*].app[0].requestLength = intWithUnit(truncnormal(350B,20B)) *# length of a request*

\*\*.client[\*].app[0].replyLength = intWithUnit(2000B) *# length of a reply*

\*\*.client[\*].app[0].numRequestsPerSession = 1 *# number of requests sent per session*

\*\*.client[\*].app[0].thinkTime = truncnormal(1s,2s) *# time gap between reuqests*

\*\*.client[\*].app[0].idleInterval = truncnormal(3s,6s) *# time gap bewtween sessions*

\*\*.client[\*].app[0].reconnectInterval = 30s *# if connection breaks, waits this much before trying to reconnect*

*# tcp client congestion control algorithm*

\*\*.client[0].tcp.tcpAlgorithmClass = "TcpTahoe"

\*\*.client[1].tcp.tcpAlgorithmClass = "TcpReno" *# default algorithm*

\*\*.client[2].tcp.tcpAlgorithmClass = "TcpNewReno"

\*\*.client[3].tcp.tcpAlgorithmClass = "TcpWestwood"

\*\*.client[4].tcp.tcpAlgorithmClass = "TcpVegas"

*# tcp server*

\*\*.server.numApps = 1

\*\*.server.tcp.typename = "Tcp"

\*\*.server.app[0].typename = "TcpGenericServerApp" *# inet.applications.tcpapp.TcpGenericServerApp*

\*\*.server.app[0].localAddress = ""

\*\*.server.app[0].localPort = 80

\*\*.server.app[0].replyDelay = 0s

*# NIC configuration (Network Initial Configuration)*

\*\*.ppp[\*].queue.typename = "DropTailQueue" *# in routers*

\*\*.ppp[\*].queue.frameCapacity = 10 *# in routers*

[Config TestUdp]

\*.n = 0

\*\*.clientUdp.numApps = 1

\*\*.client[\*].numApps = 0

\*\*.clientUdp.app[\*].sendInterval = 1s

[Config TestTcp]

\*.n = 1

\*\*.clientUdp.numApps = 0

\*\*.client[\*].numApps = 1

\*\*.client[\*].app[0].numRequestsPerSession = 1

\*\*.client[\*].app[0].replyLength = intWithUnit(1000B)

[Config TestCongestionControl]

*# Add configurations to simulate congestion environment*

3.添加配置模拟拥塞环境，可前往inet/transportlayer/tcp/flavours查看各种拥塞控制算法的源码，修改输出日志进行调试。

* 1. **示例-实验二**
     1. **ARQ实验**

1.新建ned文件，编写代码，建立两个具有收发功能的节点并连接，传输延迟设置为100ms。

其中在source中的代码如下：

simple sender

{

parameters:

@display(“i=block/process”);

gates:

input in;

output out;

}

simple reciever

{

parameters:

@display(“i=block/process”);

gates:

input in;

output out;

}

network arqnet

{

@display(“bgb=774,582”);

submodules:

sender:sender{

parameters:

@display(“i=,cyan;p=545,141”);

}

reciever: reciever {

parameters:

@display(“i=,cyan;p=545,274”);

}

connections:

sender.out --> { delay = 100ms;} --> reciever.in;

sender.in <-- { delay = 100ms;} <-- reciever.out;

}

2.添加源代码，完成停等式ARQ协议。其中接收节点程序和发送节点的程序如下：

接收端：



发送端：







3.添加ini文件，参考代码：

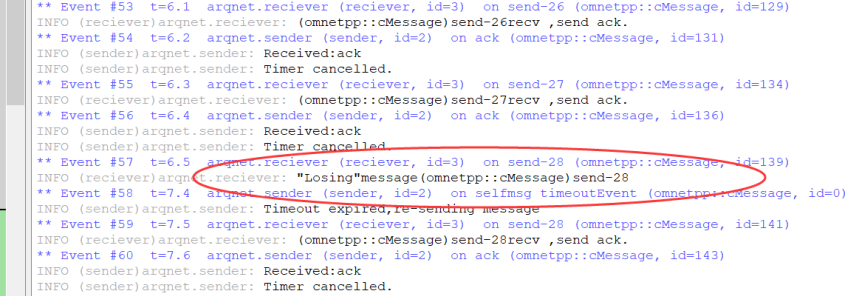
[General]

network = arqnet

sim-time-limit = 100s

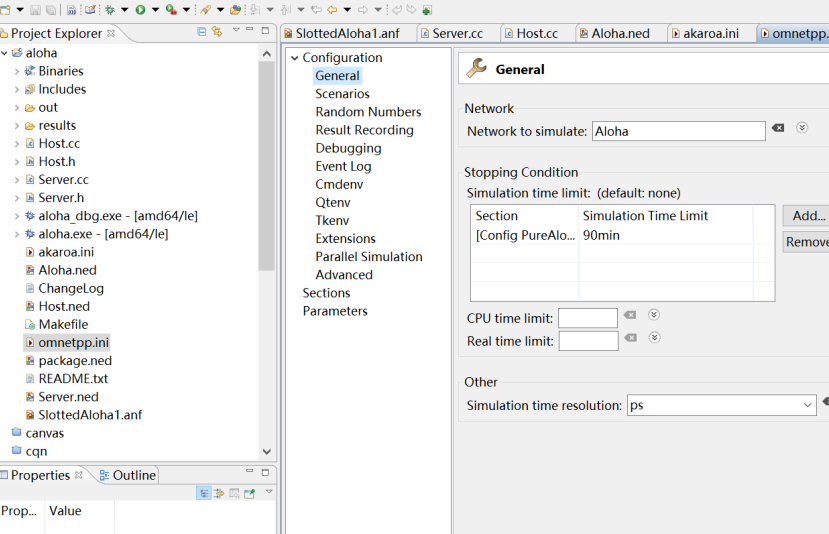
simtime-resolution = ps

4.运行仿真，并在仿真输出栏输出当前状态，例如发送失败信息：

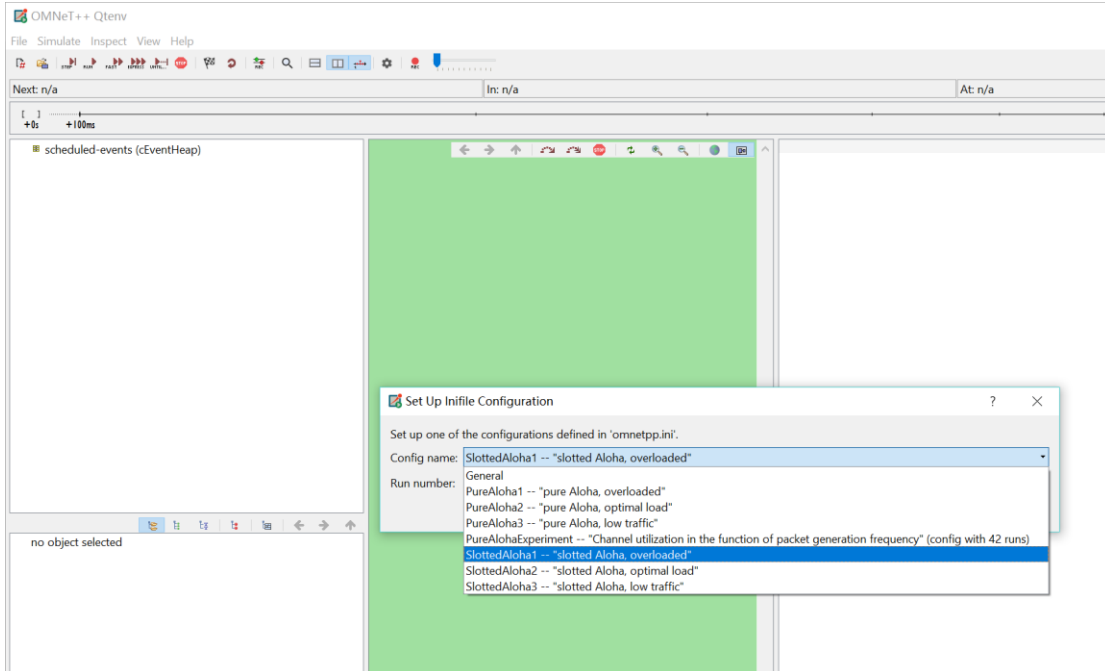


* + 1. **ALOHA实验**

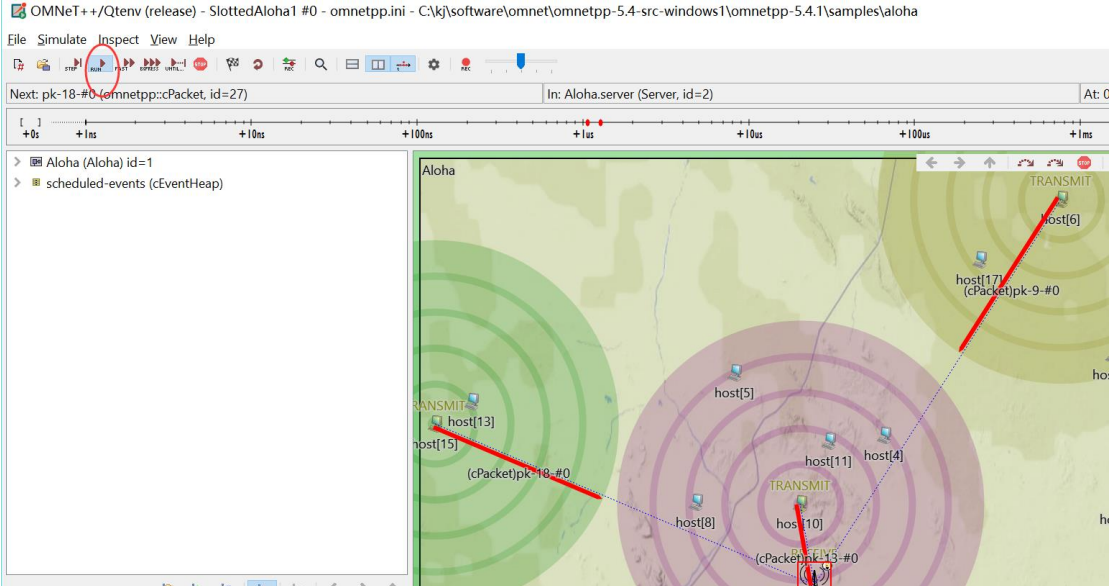
1. 在左侧文件夹中选择 aloha，双击打开文件夹，在 omnetpp.ini文件上右键选择 Run AS—>OMNet++simulation



2. 仿真窗口出现后，选择 slotted aloha。点击 ok。



3. 点击开始仿真，观察仿真波形。



* 1. **示例-实验三**
     1. **OspfRouter模块**

**#include** <map>

**#include** <memory.h>

**#include** <stdlib.h>

**#include** <string>

**#include** "inet/common/IProtocolRegistrationListener.h"

**#include** "inet/common/ModuleAccess.h"

**#include** "inet/common/lifecycle/ModuleOperations.h"

**#include** "inet/common/lifecycle/NodeStatus.h"

**#include** "inet/routing/ospfv2/Ospf.h"

**#include** "inet/routing/ospfv2/OspfConfigReader.h"

**#include** "inet/routing/ospfv2/messagehandler/MessageHandler.h"

**namespace** inet {

**namespace** ospf {

Define\_Module(Ospf);

**Ospf::Ospf**()

{

}

**Ospf::~Ospf**()

{

**cancelAndDelete**(startupTimer);

**delete** ospfRouter;

}

**void** **Ospf::initialize**(**int** stage)

{

RoutingProtocolBase::initialize(stage);

**if** (stage == *INITSTAGE\_LOCAL*) {

host = getContainingNode(**this**);

ift = getModuleFromPar<IInterfaceTable>(**par**("interfaceTableModule"), **this**);

rt = getModuleFromPar<IIpv4RoutingTable>(**par**("routingTableModule"), **this**);

startupTimer = **new** cMessage("OSPF-startup");

}

**else** **if** (stage == *INITSTAGE\_ROUTING\_PROTOCOLS*) { // interfaces and static routes are already initialized

registerService(Protocol::*ospf*, **nullptr**, **gate**("ipIn"));

registerProtocol(Protocol::*ospf*, **gate**("ipOut"), **nullptr**);

}

}

**void** **Ospf::handleMessageWhenUp**(cMessage \*msg)

{

**if** (msg == startupTimer) {

createOspfRouter();

subscribe();

}

**else**

ospfRouter->getMessageHandler()->messageReceived(msg);

}

**void** **Ospf::createOspfRouter**()

{

ospfRouter = **new** Router(**this**, ift, rt);

// read the OSPF AS configuration

cXMLElement \*ospfConfig = **par**("ospfConfig");

OspfConfigReader configReader(**this**, ift);

**if** (!configReader.loadConfigFromXML(ospfConfig, ospfRouter))

**throw** cRuntimeError("Error reading AS configuration from %s", ospfConfig->**getSourceLocation**());

ospfRouter->addWatches();

}

**void** **Ospf::subscribe**()

{

host->**subscribe**(interfaceCreatedSignal, **this**);

host->**subscribe**(interfaceDeletedSignal, **this**);

host->**subscribe**(interfaceStateChangedSignal, **this**);

}

**void** **Ospf::unsubscribe**()

{

host->**unsubscribe**(interfaceCreatedSignal, **this**);

host->**unsubscribe**(interfaceDeletedSignal, **this**);

host->**unsubscribe**(interfaceStateChangedSignal, **this**);

}

/\*\*

\* Listen on interface changes and update private data structures.

\*/

**void** **Ospf::receiveSignal**(cComponent \*source, simsignal\_t signalID, cObject \*obj, cObject \*details)

{

Enter\_Method("Ospf::receiveSignal");

**const** InterfaceEntry \*ie;

**const** InterfaceEntryChangeDetails \*change;

**if** (signalID == interfaceCreatedSignal) {

// configure interface for RIP

ie = check\_and\_cast<**const** InterfaceEntry \*>(obj);

**if** (ie->isMulticast() && !ie->isLoopback()) {

// **TODO**

}

}

**else** **if** (signalID == interfaceDeletedSignal) {

ie = check\_and\_cast<**const** InterfaceEntry \*>(obj);

// **TODO**

}

**else** **if** (signalID == interfaceStateChangedSignal) {

change = check\_and\_cast<**const** InterfaceEntryChangeDetails \*>(obj);

**if** (change->getFieldId() == InterfaceEntry::*F\_CARRIER* || change->getFieldId() == InterfaceEntry::*F\_STATE*) {

ie = change->getInterfaceEntry();

**if** (!ie->isUp())

handleInterfaceDown(ie);

**else** {

// interface went back online. Do nothing!

// Wait for Hello messages to establish adjacency.

}

}

}

**else**

**throw** cRuntimeError("Unexpected signal: %s", **getSignalName**(signalID));

}

**void** **Ospf::handleStartOperation**(LifecycleOperation \*operation)

{

ASSERT(ospfRouter == **nullptr**);

simtime\_t startupTime = **par**("startupTime");

**if** (startupTime <= simTime()) {

createOspfRouter();

subscribe();

}

**else**

**scheduleAt**(simTime() + startupTime, startupTimer);

}

**void** **Ospf::handleStopOperation**(LifecycleOperation \*operation)

{

ASSERT(ospfRouter);

**delete** ospfRouter;

**cancelEvent**(startupTimer);

ospfRouter = **nullptr**;

unsubscribe();

}

**void** **Ospf::handleCrashOperation**(LifecycleOperation \*operation)

{

ASSERT(ospfRouter);

**delete** ospfRouter;

**cancelEvent**(startupTimer);

ospfRouter = **nullptr**;

unsubscribe();

}

**void** **Ospf::insertExternalRoute**(**int** ifIndex, **const** Ipv4AddressRange& netAddr)

{

Enter\_Method\_Silent();

OspfAsExternalLsaContents newExternalContents;

newExternalContents.setRouteCost(OSPF\_BGP\_DEFAULT\_COST);

newExternalContents.setExternalRouteTag(OSPF\_EXTERNAL\_ROUTES\_LEARNED\_BY\_BGP);

**const** Ipv4Address netmask = netAddr.mask;

newExternalContents.setNetworkMask(netmask);

ospfRouter->updateExternalRoute(netAddr.address, newExternalContents, ifIndex);

}

**int** **Ospf::checkExternalRoute**(**const** Ipv4Address& route)

{

Enter\_Method\_Silent();

**for** (uint32\_t i = 0; i < ospfRouter->getASExternalLSACount(); i++) {

AsExternalLsa \*externalLSA = ospfRouter->getASExternalLSA(i);

Ipv4Address externalAddr = externalLSA->getHeader().getLinkStateID();

**if** (externalAddr == route) { //**FIXME** was this meant???

**if**(externalLSA->getContents().getE\_ExternalMetricType())

**return** 2;

**else**

**return** 1;

}

}

**return** 0;

}

**void** **Ospf::handleInterfaceDown**(**const** InterfaceEntry \*ie)

{

EV\_DEBUG << "interface " << ie->getInterfaceId() << " went down. \n";

// Step 1: delete all direct-routes connected to this interface

// ... from OSPF table

**for**(uint32\_t i = 0; i < ospfRouter->getRoutingTableEntryCount(); i++) {

OspfRoutingTableEntry \*ospfRoute = ospfRouter->getRoutingTableEntry(i);

**if**(ospfRoute && ospfRoute->getInterface() == ie && ospfRoute->getNextHopAsGeneric().isUnspecified()) {

EV\_DEBUG << "removing route from OSPF routing table: " << ospfRoute << "\n";

ospfRouter->deleteRoute(ospfRoute);

i--;

}

}

// ... from Ipv4 table

**for**(int32\_t i = 0; i < rt->getNumRoutes(); i++) {

Ipv4Route \*route = rt->getRoute(i);

**if**(route && route->getInterface() == ie && route->getNextHopAsGeneric().isUnspecified()) {

EV\_DEBUG << "removing route from Ipv4 routing table: " << route << "\n";

rt->deleteRoute(route);

i--;

}

}

// Step 2: find the OspfInterface associated with the ie and take it down

OspfInterface \*foundIntf = **nullptr**;

**for**(**auto** &areaId : ospfRouter->getAreaIds()) {

Area \*area = ospfRouter->getAreaByID(areaId);

**if**(area) {

**for**(**auto** &ifIndex : area->getInterfaceIndices()) {

OspfInterface \*intf = area->getInterface(ifIndex);

**if**(intf && intf->getIfIndex() == ie->getInterfaceId()) {

foundIntf = intf;

**break**;

}

}

**if**(foundIntf) {

foundIntf->processEvent(OspfInterface::*INTERFACE\_DOWN*);

**break**;

}

}

}

}

} // namespace ospf

} // namespace inet

* + 1. **RipRouter模块**

**void** **Rip::handleMessageWhenUp**(cMessage \*msg)

{

**if** (msg->isSelfMessage()) {

**if** (msg == updateTimer) {

processUpdate(**false**);

**scheduleAt**(simTime() + updateInterval, msg);

}

**else** **if** (msg == triggeredUpdateTimer) {

processUpdate(**true**);

}

**else** **if** (msg == startupTimer) {

startRIPRouting();

}

**else** **if** (msg == shutdownTimer) {

ASSERT(operationalState == State::*STOPPING\_OPERATION*);

finishActiveOperation();

}

**else**

**throw** cRuntimeError("unknown self message");

}

**else** **if** (msg->getKind() == *UDP\_I\_DATA*) {

Packet \*pk = check\_and\_cast<Packet \*>(msg);

**unsigned** **char** command = pk->peekAtFront<RipPacket>()->getCommand();

**if** (command == *RIP\_REQUEST*)

processRequest(pk);

**else** **if** (command == *RIP\_RESPONSE*)

processResponse(pk);

**else**

**throw** cRuntimeError("RIP: unknown command (%d)", (**int**)command);

}

**else** **if** (msg->getKind() == *UDP\_I\_ERROR*) {

EV\_DETAIL << "Ignoring UDP error report\n";

**delete** msg;

}

}

**void** **Rip::startRIPRouting**()

{

addressType = rt->getRouterIdAsGeneric().getAddressType();

cXMLElementList interfaceElements = **par**("ripConfig").**xmlValue**()->**getChildrenByTagName**("interface");

InterfaceMatcher matcher(interfaceElements);

// Creates a RipInterfaceEntry for each interface found in the 'interface table'.

**for** (**int** k = 0; k < ift->getNumInterfaces(); ++k) {

InterfaceEntry \*ie = ift->getInterface(k);

**if** (ie->isMulticast() && !ie->isLoopback()) {

**int** i = matcher.findMatchingSelector(ie);

addRipInterface(ie, i >= 0 ? interfaceElements[i] : **nullptr**);

}

}

// Import interface/static/default routes from the 'routing table'.

**for** (**int** i = 0; i < rt->getNumRoutes(); ++i) {

IRoute \*route = rt->getRoute(i);

**if** (isLoopbackInterfaceRoute(route)) {

/\*ignore\*/

;

}

**else** **if** (isLocalInterfaceRoute(route)) {

InterfaceEntry \*ie = check\_and\_cast<InterfaceEntry \*>(route->getSource());

RipInterfaceEntry \*ripIe = findRipInterfaceById(ie->getInterfaceId());

**if**(!ripIe || ripIe->mode != *NO\_RIP*)

importRoute(route, RipRoute::*RIP\_ROUTE\_INTERFACE*, getInterfaceMetric(ie));

}

**else** **if** (isDefaultRoute(route))

importRoute(route, RipRoute::*RIP\_ROUTE\_DEFAULT*);

**else** {

**const** L3Address& destAddr = route->getDestinationAsGeneric();

**if** (!destAddr.isMulticast() && !destAddr.isLinkLocal())

importRoute(route, RipRoute::*RIP\_ROUTE\_STATIC*);

}

}

// subscribe to interface created/deleted/changed notifications

host->**subscribe**(interfaceCreatedSignal, **this**);

host->**subscribe**(interfaceDeletedSignal, **this**);

host->**subscribe**(interfaceStateChangedSignal, **this**);

// subscribe to route added/deleted/changed notifications

host->**subscribe**(routeAddedSignal, **this**);

host->**subscribe**(routeDeletedSignal, **this**);

host->**subscribe**(routeChangedSignal, **this**);

// configure socket

socket.setMulticastLoop(**false**);

socket.bind(ripUdpPort);

**for** (**auto** & elem : ripInterfaces)

**if** (elem.mode != *NO\_RIP* && elem.mode != *PASSIVE*)

socket.joinMulticastGroup(addressType->getLinkLocalRIPRoutersMulticastAddress(), elem.ie->getInterfaceId());

**for** (**auto** & elem : ripInterfaces)

**if** (elem.mode != *NO\_RIP* && elem.mode != *PASSIVE*)

sendRIPRequest(elem);

// set update timer

**scheduleAt**(simTime() + updateInterval, updateTimer);

}

**void** **Rip::stopRIPRouting**()

{

**if** (startupTimer->isScheduled())

**cancelEvent**(startupTimer);

**else** {

socket.close();

// unsubscribe to notifications

host->**unsubscribe**(interfaceCreatedSignal, **this**);

host->**unsubscribe**(interfaceDeletedSignal, **this**);

host->**unsubscribe**(interfaceStateChangedSignal, **this**);

host->**unsubscribe**(routeDeletedSignal, **this**);

host->**unsubscribe**(routeAddedSignal, **this**);

host->**unsubscribe**(routeChangedSignal, **this**);

}

// cancel timers

**cancelEvent**(updateTimer);

**cancelEvent**(triggeredUpdateTimer);

// clear data

**for** (**auto**& elem : ripRoutingTable)

**delete** elem;

ripRoutingTable.clear();

ripInterfaces.clear();

}

/\*\*

\* Listen on interface/route changes and update private data structures.

\*/

**void** **Rip::receiveSignal**(cComponent \*source, simsignal\_t signalID, cObject \*obj, cObject \*details)

{

Enter\_Method\_Silent("Rip::receiveChangeNotification(%s)", cComponent::**getSignalName**(signalID));

**const** InterfaceEntry \*ie;

**const** InterfaceEntryChangeDetails \*change;

**if** (signalID == interfaceCreatedSignal) {

// configure interface for RIP

ie = check\_and\_cast<**const** InterfaceEntry \*>(obj);

**if** (ie->isMulticast() && !ie->isLoopback()) {

cXMLElementList config = **par**("ripConfig").**xmlValue**()->**getChildrenByTagName**("interface");

**int** i = InterfaceMatcher(config).findMatchingSelector(ie);

**if** (i >= 0)

addRipInterface(ie, config[i]);

}

}

**else** **if** (signalID == interfaceDeletedSignal) {

// delete interfaces and routes referencing the deleted interface

ie = check\_and\_cast<**const** InterfaceEntry \*>(obj);

deleteRipInterface(ie);

}

**else** **if** (signalID == interfaceStateChangedSignal) {

change = check\_and\_cast<**const** InterfaceEntryChangeDetails \*>(obj);

**if** (change->getFieldId() == InterfaceEntry::*F\_CARRIER* || change->getFieldId() == InterfaceEntry::*F\_STATE*) {

ie = change->getInterfaceEntry();

**if** (!ie->isUp()) {

**for** (**auto** & elem : ripRoutingTable)

**if** ((elem)->getInterface() == ie) {

invalidateRoute(elem);

}

}

**else** {

RipInterfaceEntry \*ripInterfacePtr = findRipInterfaceById(ie->getInterfaceId());

**if** (ripInterfacePtr && ripInterfacePtr->mode != *NO\_RIP* && ripInterfacePtr->mode != *PASSIVE*)

sendRIPRequest(\*ripInterfacePtr);

}

}

}

**else** **if** (signalID == routeDeletedSignal) {

// remove references to the deleted route and invalidate the RIP route

**const** IRoute \*route = check\_and\_cast<**const** IRoute \*>(obj);

**for** (**auto** & elem : ripRoutingTable) {

**if** ((elem)->getRoute() == route) {

(elem)->setRoute(**nullptr**);

**if** (route->getSource() != **this**) {

invalidateRoute(elem);

}

}

}

}

**else** **if** (signalID == routeAddedSignal) {

// add or update the RIP route

IRoute \*route = **const\_cast**<IRoute \*>(check\_and\_cast<**const** IRoute \*>(obj));

**if** (route->getSource() != **this**) {

**if** (isLoopbackInterfaceRoute(route)) {

/\*ignore\*/

;

}

**else** **if** (isLocalInterfaceRoute(route)) {

InterfaceEntry \*ie = check\_and\_cast<InterfaceEntry \*>(route->getSource());

RipRoute \*ripRoute = findRipRoute(ie, RipRoute::*RIP\_ROUTE\_INTERFACE*);

**if** (ripRoute) { // readded

RipInterfaceEntry \*ripIe = findRipInterfaceById(ie->getInterfaceId());

ripRoute->setRoute(route);

ripRoute->setMetric(ripIe ? ripIe->metric : 1);

ripRoute->setChanged(**true**);

triggerUpdate();

}

**else**

{

RipInterfaceEntry \*ripIe = findRipInterfaceById(ie->getInterfaceId());

**if**(!ripIe || ripIe->mode != *NO\_RIP*)

importRoute(route, RipRoute::*RIP\_ROUTE\_INTERFACE*, getInterfaceMetric(ie));

}

}

**else** {

// **TODO** import external routes from other routing daemons

}

}

}

**else** **if** (signalID == routeChangedSignal) {

**const** IRoute \*route = check\_and\_cast<**const** IRoute \*>(obj);

**if** (route->getSource() != **this**) {

RipRoute \*ripRoute = findRipRoute(route);

**if** (ripRoute) {

// **TODO** check and update tag

**bool** changed = route->getDestinationAsGeneric() != ripRoute->getDestination() ||

route->getPrefixLength() != ripRoute->getPrefixLength() ||

route->getNextHopAsGeneric() != ripRoute->getNextHop() ||

route->getInterface() != ripRoute->getInterface();

ripRoute->setDestination(route->getDestinationAsGeneric());

ripRoute->setPrefixLength(route->getPrefixLength());

ripRoute->setNextHop(route->getNextHopAsGeneric());

ripRoute->setInterface(route->getInterface());

**if** (changed) {

ripRoute->setChanged(**true**);

triggerUpdate();

}

}

}

}

**else**

**throw** cRuntimeError("Unexpected signal: %s", **getSignalName**(signalID));

}

* + 1. **BgpRouter模块**

**void** **Bgp::initialize**(**int** stage)

{

cSimpleModule::initialize(stage);

**if** (stage == *INITSTAGE\_LOCAL*) {

ift = getModuleFromPar<IInterfaceTable>(**par**("interfaceTableModule"), **this**);

rt = getModuleFromPar<IIpv4RoutingTable>(**par**("routingTableModule"), **this**);

startupTimer = **new** cMessage("BGP-startup");

}

**else** **if** (stage == *INITSTAGE\_ROUTING\_PROTOCOLS*) { // interfaces and static routes are already initialized

cModule \*node = findContainingNode(**this**);

NodeStatus \*nodeStatus = node ? check\_and\_cast\_nullable<NodeStatus \*>(node->**getSubmodule**("status")) : **nullptr**;

isUp = !nodeStatus || nodeStatus->getState() == NodeStatus::*UP*;

**if** (isUp) {

simtime\_t startupTime = **par**("startupTime");

**if** (startupTime == 0)

createBgpRouter();

**else**

**scheduleAt**(simTime() + startupTime, startupTimer);

}

}

}

**void** **Bgp::finish**()

{

**if** (!isUp)

{

EV\_ERROR << "Protocol is turned off. \n";

**return**;

}

bgpRouter->recordStatistics();

}

**void** **Bgp::handleMessage**(cMessage \*msg)

{

**if** (!isUp)

{

**if** (msg->isSelfMessage())

**throw** cRuntimeError("Model error: self msg '%s' received when protocol is down", msg->getName());

EV\_ERROR << "Protocol is turned off, dropping '" << msg->getName() << "' message\n";

**delete** msg;

**return**;

}

**if**(msg == startupTimer)

createBgpRouter();

**else** **if** (msg->isSelfMessage()) // BGP level

handleTimer(msg);

**else** **if** (!**strcmp**(msg->**getArrivalGate**()->**getName**(), "socketIn")) // TCP level

bgpRouter->processMessageFromTCP(msg);

**else**

**delete** msg;

}

**void** **Bgp::createBgpRouter**()

{

bgpRouter = **new** BgpRouter(**this**, ift, rt);

// read BGP configuration

cXMLElement \*bgpConfig = **par**("bgpConfig");

BgpConfigReader configReader(**this**, ift);

configReader.loadConfigFromXML(bgpConfig, bgpRouter);

bgpRouter->printSessionSummary();

bgpRouter->addWatches();

}

**void** **Bgp::handleTimer**(cMessage \*timer)

{

BgpSession \*pSession = (BgpSession \*)timer->getContextPointer();

**if** (pSession) {

**switch** (timer->getKind()) {

**case** START\_EVENT\_KIND:

EV\_INFO << "Processing Start Event" << std::**endl**;

pSession->getFSM()->ManualStart();

**break**;

**case** CONNECT\_RETRY\_KIND:

EV\_INFO << "Expiring Connect Retry Timer" << std::**endl**;

pSession->getFSM()->ConnectRetryTimer\_Expires();

**break**;

**case** HOLD\_TIME\_KIND:

EV\_INFO << "Expiring Hold Timer" << std::**endl**;

pSession->getFSM()->HoldTimer\_Expires();

**break**;

**case** KEEP\_ALIVE\_KIND:

EV\_INFO << "Expiring Keep Alive timer" << std::**endl**;

pSession->getFSM()->KeepaliveTimer\_Expires();

**break**;

**default**:

**throw** cRuntimeError("Invalid timer kind %d", timer->getKind());

}

}

}

* 1. **示例-实验四**
     1. **组网模块**

1.添加ned文件建立拓扑，部分代码示例如下（并不是实验的实际拓扑）：

**network** Network

{

**types**:

**channel** LINK\_100 **extends** ThruputMeteringChannel

{

**parameters**:

delay = 50us;

datarate = 100Mbps;

displayAsTooltip = true;

thruputDisplayFormat = "#N";

}

**submodules**:

visualizer: IntegratedCanvasVisualizer {

**parameters**:

**@display**("p=100,100;is=s");

}

configurator: Ipv4NetworkConfigurator {

**parameters**:

**@display**("p=100,200;is=s");

config = **xmldoc**("IPv4Config.xml");

addStaticRoutes = false;

addDefaultRoutes = false;

addSubnetRoutes = false;

}

A: Router {

**parameters**:

**@display**("p=700,230");

**gates**:

pppg[3];

}

B: Router {

**parameters**:

**@display**("p=850,230");

**gates**:

pppg[3];

}

**connections:**

A.pppg[0] <--> LINK\_100 <--> B.pppg[0];

}

2.为各接口分配IP，其中配置ip的xml文件如下：

<config>

<interface hosts='A' names='ppp0' address='100.10.10.1' netmask='255.255.255.0' groups='224.0.0.1 224.0.0.2 224.0.0.5' metric='1'/>

<interface hosts='A' names='ppp1' address='10.10.1.2' netmask='255.255.255.0' groups='224.0.0.1 224.0.0.2 224.0.0.5' metric='1'/>

<interface hosts='A' names='ppp2' address='10.10.3.2' netmask='255.255.255.0' groups='224.0.0.1 224.0.0.2 224.0.0.5' metric='1'/>

<interface hosts='B' names='ppp0' address='100.10.10.2' netmask='255.255.255.0' groups='224.0.0.1 224.0.0.2 224.0.0.5' metric='1'/>

<interface hosts='B' names='ppp1' address='10.20.1.2' netmask='255.255.255.0' groups='224.0.0.1 224.0.0.2 224.0.0.5' metric='1'/>

<interface hosts='B' names='ppp2' address='10.20.3.2' netmask='255.255.255.0' groups='224.0.0.1 224.0.0.2 224.0.0.5' metric='1'/>

<config>